Proyecto Fin de Máster.
Curso 2009-2010.

Extensión de la Política de Reemplazamiento peLifo a Entornos Multicore

Autor:
Silvio Sepúlveda

Directores del proyecto:
Daniel Ángel Chaver Martínez
Fernando Castro Rodríguez

Facultad de Informática.
Universidad Complutense de Madrid.

Autorización

El abajo firmante, matriculado/a en el Máster en Investigación en Informática de la Facultad de Informática, autoriza a la Universidad Complutense de Madrid (UCM) a difundir y utilizar con fines académicos, no comerciales y mencionando expresamente a su autor el presente Trabajo Fin de Máster: "Extensión de la política de reemplazamiento Probabilistic Escape LIFO a entornos multicore", realizado durante el curso académico 2009-2010 bajo la dirección de Daniel Ángel Chaver Martínez y Fernando Castro Rodríguez en el Departamento de Arquitectura de Computadores y Automática, y a la Biblioteca de la UCM a depositarlo en el Archivo Institucional E-Prints Complutense con el objeto de incrementar la difusión, uso e impacto del trabajo en Internet y garantizar su preservación y acceso a largo plazo.

Silvio Sepúlveda

Proyecto Fin de Máster.
Curso 2009-2010.

Extensión de la Política de Reemplazamiento peLifo a Entornos Multicore

Autor:
Silvio Sepúlveda

Directores del proyecto:
Daniel Ángel Chaver Martínez
Fernando Castro Rodríguez

Facultad de Informática.
Universidad Complutense de Madrid.

Índice general

Índice de figuras

Índice de tablas

Resumen

En la actualidad, continúa el incremento entre la diferencia de velocidad del procesador y la memoria. Esta diferencia de velocidades ha sido mitigada en forma parcial mediante el uso de la jerarquía de memoria que se construye usando diferentes tipos de memoria, logrando que la velocidad de acceso se acerque al componente más rápido y la capacidad al componente más grande.

Con la llegada de los sistemas *multicore*, algunos de los aspectos de diseño deben ser replanteados para sacar el máximo provecho a este tipo de sistemas. Uno de estos aspectos, es el manejo de la cache de último nivel (LLC) *on-chip*, pues ésta es la frontera antes de pasar a la memoria principal, la cual presenta una latencia de acceso mucho más alta. Una de las formas de mejorar el comportamiento de la LLC, es mediante la modificación de la política de reemplazamiento.

Este trabajo se centra en la política de reemplazamiento *probabilistic escape lifo* (peLifo). Esta política ha presentado buenos resultados, en términos de la tasa de fallos obtenida, en entornos *singlecore*, por tanto el objetivo de este trabajo es ampliar la política a entornos *multicore*, para intentar sacarle mayor provecho.

Tomando en cuenta este objetivo, se han planteado una serie de modifica-

ciones a la política peLifo original, agregándole información sobre el comportamiento individual de cada uno de los procesos que se encuentren en ejecución en un momento dado. De este modo, se pretende privilegiar a los procesos que realizan un mejor uso de la LLC conservando sus bloques y penalizar a los otros procesos reemplazando sus bloques.

Abstract

Today, the difference in the speed of processor and memory continues to increase. This difference has been partially mitigated building a memory hierarchy that uses different types of memory, making the access time to approach to the fastest component and the capacity to the biggest component.

With the arrival of multicore systems, some of the design considerations must be adapted to get more benefits out of these systems. One of those aspects is the on-chip last level cache (LLC) management, since a miss on that level implies accessing main memory, which has considerable latency. One of the ways to improve the LLC behavior is modifying the replacement policy.

This work focuses in the probabilistic escape lifo (peLifo) replacement policy, that obtains very low miss rates for singlecore systems. In this work we try to extend this policy to multicore systems to get even more benefit of it.

With this objective in mind, this work proposes several modifications to the original peLifo policy. All of them, add information about each executing process, and use it to keep the blocks of those processes that make a better use of the LLC, and to penalize the others by evicting their blocks.

Palabras clave

- Cache
- Políticas de reemplazamiento
- Probabilistic Escape LIFO
- Memoria
- Multiprocesador
- Rendimiento
- LRU

Keywords

- Cache
- Replacement policies
- Probabilistic Escape LIFO
- Memory
- Multicore
- Performance
- LRU

Capítulo 1

Introducción

En esta sección se describe el problema de la velocidad limitada de la memoria y cómo afecta al rendimiento de un sistema de cómputo, además de algunas soluciones que se han propuesto para solucionar dicho problema. A continuación se define la motivación y los objetivos específicos del presente trabajo, y se finaliza con una descripción de la organización de este documento.

1.1. El *memory gap*

La brecha cada vez mayor entre las velocidades del procesador y la memoria (Figura 1.1) ha hecho que la organización, la arquitectura y el diseño de los sistemas de memoria sean un aspecto cada vez más importante en el diseño de sistemas de cómputo. Se ha llegado al punto que los parámetros de la jerarquía de memoria afectan más al rendimiento del sistema que los parámetros del procesador [JWN07], en consecuencia es indispensable para cualquier diseñador tener un conocimiento profundo de la organización del sistema de

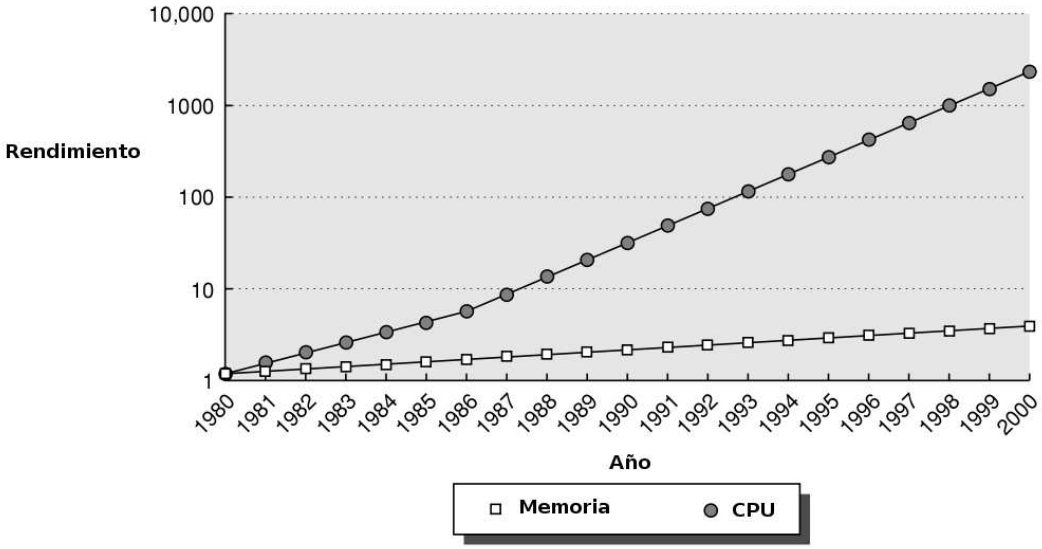

FIGURA 1.1: Diferencia de rendimiento entre la memoria y el procesador en el tiempo [HP02].

memoria, además de su operación y comportamiento.

Por otra parte, los sistemas *multicore* son cada vez más comunes, pues permiten la ejecución concurrente de varias aplicaciones. En estos sistemas es frecuente encontrar caches de último nivel (*Last Level Cache* - LLC) compartidas, ya que permiten una disposición más flexible y dinámica de los recursos. Sin embargo, tienen como inconveniente que a medida que el número de *cores* aumenta, la contención causada por las aplicaciones que comparten la LLC se incrementa también, por lo que el rendimiento de estos sistemas estará muy influenciado por la eficiencia con la que se maneja la cache compartida [JHQ+08]. Debido a esto, un aspecto clave de diseño que deberán afrontar los arquitectos de procesadores es la organización y manejo de la LLC *on-chip*, buscando reducir al máximo los accesos a memoria *off-chip*, para disminuir así

las latencias de acceso.

1.2. Jerarquía de memoria

La implementación de una jerarquía de memoria ha sido la solución adoptada para proveer una gran cantidad de memoria de alta velocidad a un bajo costo. Su diseño se basa principalmente en dos observaciones: la primera es el principio de localidad, el cual establece que si una aplicación hace referencia a una posición de memoria (que contenga instrucciones o datos), es muy probable que la vuelva a referenciar en un instante de tiempo cercano (localidad temporal), y que las ubicaciones cercanas a ella, también sean referenciadas poco después (localidad espacial). La segunda es que mientras más pequeño sea el hardware, más rápido podrá funcionar.

Con estas dos ideas, se ha diseñado la jerarquía de memoria utilizando memorias de diferentes tamaños y velocidades, lo que permite que un sistema de memoria se acerque simultáneamente al rendimiento del componente más rápido y al costo por bit del componente más barato [HP02]. De modo que, cada uno de los niveles de memoria, mapea direcciones desde los niveles más bajos, que tienen mayor cantidad de memoria, pero más lenta, a los niveles más altos, que tienen una menor cantidad de memoria, aunque con mayor velocidad.

Una jerarquía de memoria moderna está compuesta por los siguientes componentes [JWN07]:

- Cache: Es el nivel más alto en la jerarquía de memoria y el más rápido. Éste provee acceso al código y a los datos con una latencia muy baja y

un gran ancho de banda, pero a costa de ser el más costoso de toda la jerarquía.

- DRAM: Proporciona almacenamiento de acceso aleatorio el cual es relativamente grande, relativamente rápido y relativamente costoso.
- Disco: Es el último nivel de la jerarquía, es el más barato y con mayor capacidad de almacenamiento, pero también es el más lento. Normalmente se utiliza como medio de almacenamiento permanente.

En este trabajo, el componente que más nos interesa es la memoria cache, de la cual se hará una descripción detallada en la siguiente sección.

1.2.1. Memoria cache

Como se mencionó anteriormente, la cache es el nivel más alto de la jerarquía de memoria y, por tanto, el que se encuentra más cercano al procesador. Cuando el procesador necesita un dato que está en memoria, la petición llega en primer lugar a la cache, que busca si el dato pedido se encuentra o no almacenado en ella. Si encuentra el dato, se dice que se ha producido un *acierto* y el dato se hace llegar al procesador. De lo contrario, se dice que se ha generado un *fallo* y la petición se hace llegar al siguiente nivel de la jerarquía, que puede ser tanto otra memoria cache como una memoria DRAM.

Se diferencian tres tipos de fallos de cache. Los primeros se deben a que los datos nunca hayan sido referenciados, por lo que no se encuentran en la cache y se denominan *fallos iniciales*. Los segundos son debidos al tamaño limitado de la cache y son llamados *fallos de capacidad*. El último de ellos es debido a las restricciones de emplazamiento y se denominan *fallos de conflicto*.

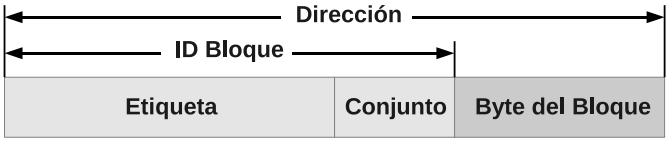

FIGURA 1.2: Organización lógica de la cache

La implementación de una memoria cache debe tomar en consideración tres aspectos clave: el ordenamiento lógico de los datos, las heurísticas de gestión de contenido y las heurísticas para mantener la consistencia.

Ordenamiento lógico

La cache almacena fragmentos de datos, denominados *bloques* o *líneas*, que vienen del nivel superior de la jerarquía de memoria. Como consecuencia de este ordenamiento (figura 1.2), se divide lógicamente el espacio de direcciones, lo que proporciona un modo simple y efectivo de identificar un bloque: el *identificador de bloque*.

Debido al tamaño finito de la cache, es necesario implementar un mecanismo para indicar si un dato se encuentra o no almacenado. Para esto se utilizan las denominadas *etiquetas*, las cuales se forman con una parte de los bits del identificador del bloque. Así, para encontrar un bloque dentro de la cache, se compara una parte de su dirección con la etiqueta almacenada.

Se definen también agrupaciones de bloques que determinan en qué partes de la cache puede almacenarse un determinado bloque de datos. A cada una de estas agrupaciones se le da el nombre de *conjunto*. Estos conjuntos quedan definidos también mediante un subconjunto de bits tomados de su identificador de bloque. Existen tres organizaciones básicas de la cache que resultan de la definición de los conjuntos:

- Mapeo directo: En este tipo de organización, cada bloque sólo tiene un sitio en la cache donde puede ser almacenado, por lo que ésta tiene tantos conjuntos como bloques.

- Totalmente asociativa: En este caso, un bloque puede ser almacenado en cualquier sitio de la cache, es decir, se define sólo un conjunto, por lo que el campo de la etiqueta ocupa todo el identificador de bloque.

- Asociativa por conjuntos: Este tipo de cache es un punto medio entre los casos anteriores, por lo que tiene más de un conjunto y cada bloque de datos puede almacenarse en cualquier sitio dentro del conjunto que le corresponda.

Heurísticas de gestión de contenido

Estas heurísticas definen qué elementos se almacenarán o no en la cache en un momento determinado de la ejecución. Generalmente, se implementan a nivel de hardware y se pueden basar en cualquier característica que pueda definir a un elemento de datos: quién lo usa, cómo se usa, qué tan importante es, etc. De modo que, con esta información, se define si se almacena o no el elemento.

En [Bel66] Belady determina un mínimo teórico para el número de fallos que se puede alcanzar; su propuesta desaloja el bloque que es accedido más lejos en el tiempo, para esto utiliza un oráculo para la escogencia del bloque a reemplazar. Sin embargo, su propuesta no es implementable pues, en general, no se conoce de antemano el comportamiento futuro de la aplicación.

A las heurísticas utilizadas para decidir qué bloque será desalojado se les llama políticas de reemplazamiento. En general, estas políticas intentan acer-

carse al mínimo determinado por Belady, pero con un costo de implementación razonable. Las políticas de reemplazamiento más básicas son:

- Menos recientemente utilizado (*Least Recently Used* - LRU): Basándose en la localidad temporal, esta política considera que el bloque más apto para ser reemplazado es el bloque que lleva más tiempo sin ser utilizado.
- Menos frecuentemente utilizado (*Least Frequently Used* - LFU): Esta política reemplaza el bloque que ha sido referenciado una menor cantidad de veces.
- Primero en entrar, primero en salir (*First In, First Out* - FIFO): Debido a que la política LRU puede ser costosa de implementar, se puede aproximar con FIFO. Dicha política reemplaza el bloque que ha ingresado hace más tiempo en la cache, en lugar del menos recientemente utilizado.
- Aleatorio: Esta política selecciona aleatoriamente el bloque a reemplazar.

La política más ampliamente utilizada es LRU, pero ésta presenta algunos inconvenientes. Uno de sus principales problemas es que si un bloque que ingresa a la cache no va a volver a ser referenciado, éste debe pasar por todas las posiciones de la cola antes de ser reemplazado, lo que ocasiona que el uso de una de las vías de la cache se desperdicie en ese bloque que no genera ningún beneficio. Además, si la aplicación trabaja de forma iterativa sobre un conjunto de N datos y el tamaño de los conjuntos de la cache involucrados es $M < N$, LRU generará N fallos en cada iteración.

Otro problema se manifiesta cuando se aplica LRU en entornos *multicore*, pues asigna los recursos de la cache a las aplicaciones en ejecución basándose

únicamente en la demanda, lo que puede hacer que le sean asignados recursos a una aplicación que no los aproveche lo suficiente, es decir, una sola aplicación puede fácilmente polucionar la cache con sus datos, lo cual puede causar tasas de fallos más altas en las demás aplicaciones, causando así un rendimiento medio más bajo [SRD04].

Heurísticas de administración de consistencia

Estas heurísticas se aseguran que el código y los datos que recibe la aplicación software sean los adecuados, por lo que se ocupan principalmente de tres aspectos:

- Mantener la cache consistente consigo misma, es decir, que nunca se tendrán dos copias de un mismo dato en la cache, a no ser que se garantice que en todo momento estos dos valores son iguales.
- Mantener la cache consistente con los niveles superiores de la jerarquía, es decir, que la copia del valor en los niveles superiores debe estar actualizada con el valor de la cache, de modo que se asegura que una lectura de éstos siempre obtendrá el valor más reciente.
- Mantener la cache consistente con otras caches en caso que se tengan múltiples caches en un mismo nivel, de modo que si se tienen dos copias separadas de un dato, los valores de éstos se mantengan actualizados.

1.3. Motivación y objetivos

Debido a que los entornos multiprocesador con memoria compartida son cada vez más comunes, se hace necesaria la adaptación de las diferentes partes

del sistema a estos entornos. En particular, es de especial interés la adaptación de la jerarquía de memoria, ya que es una de las partes que más influye en el rendimiento global del sistema.

Es por esto que en este trabajo se pretende realizar una propuesta que permita adaptar una de las políticas de reemplazamiento de la cache a este tipo de entornos, haciendo que se puedan aprovechar mejor los recursos disponibles.

La política de reemplazamiento peLifo [Cha09b], tal como fue implementada originalmente, no toma en consideración el comportamiento individual de cada uno de los procesos que se encuentran en ejecución en un momento determinado. En este trabajo, se pretende realizar una extensión de ésta política, con el fin de mejorar su rendimiento en entornos multitarea y *multicore*.

1.4. Organización del documento

La organización general de este documento es como sigue:

- El capítulo 2 hace un recuento de los principales trabajos realizados en esta área.
- El capítulo 3 describe cada una de las propuestas realizadas en este trabajo para cumplir con el objetivo propuesto.
- En el capítulo 4 se describen las diferentes herramientas utilizadas en el desarrollo del trabajo, así como el simulador que fue parcialmente implementado.
- El capítulo 5 presenta los resultados obtenidos al evaluar las propuestas realizadas en este trabajo.

- Por último, el capítulo 6 presenta las conclusiones obtenidas y el trabajo futuro que podría resultar de este estudio.

Capítulo 2

Trabajo relacionado

Existe una gran cantidad de investigación relacionada con este tema, por lo que en esta sección sólo se hará una breve presentación de algunas de las principales propuestas realizadas, incluyendo aquella en la que se basa este trabajo.

2.1. Políticas *singlecore*

Esta sección se centra en las políticas que han sido concebidas para entornos monoprocesador, por lo que consideran que en cada momento sólo se encuentra un proceso en ejecución y, por tanto, sólo un proceso accede a la memoria en un determinado instante de tiempo.

2.1.1. Políticas de reemplazamiento basadas en predicción de bloques muertos

La idea principal es identificar los bloques muertos en la cache, es decir, los bloques que ya no volverán a ser referenciados antes de ser desalojados, de modo que sean los primeros en ser reemplazados, para así utilizar los recursos de la cache de manera más eficiente.

La primera propuesta que utilizó esta técnica fue realizada por Lai [LFF01], en ella se propone un predictor de bloques muertos (*Dead Block Predictor* - DBP) basado en trazas, que predice cuándo un bloque en la cache L1 de datos se puede considerar muerto.

Para esto, genera una tabla de historia duplicando el *array* de etiquetas de la cache L1 y almacena una traza de instrucciones codificada asociada a cada etiqueta. Cada traza es codificada usando adición truncada.

Además, implementa una tabla de bloques muertos que mantiene las trazas que terminan con un bloque muerto ya codificadas, a las que denomina *signatures*. Cuando llega una nueva referencia a un bloque, se busca la traza en la tabla de historia, se codifica y, por último, el DBP busca en la tabla de bloques muertos una coincidencia entre la traza que ingresa y una *signature* previamente almacenada para generar una predicción.

Una propuesta más reciente, realizada por Liu [LFHB08], predice los bloques muertos basándose en ráfagas de accesos a un bloque de cache, ya que, según su trabajo, la historia de referencias individuales de un bloque en la cache L1 puede ser irregular debido a dependencias de datos y de control.

El concepto de ráfaga en la cache es definido como el grupo contiguo de

accesos a un bloque cuando se encuentra en la posición más recientemente utilizada (*Most Recently Used* - MRU) de su conjunto sin que se produzcan referencias a otro bloque del mismo conjunto, es decir, la ráfaga termina cuando el bloque deja la posición MRU. Basándose en este concepto, proponen dos predictores. El primero, denominado predictor de conteo de ráfagas, lleva la cuenta de las ráfagas que tiene un bloque para definir cuando muere. El segundo, al que llaman predictor de trazas de ráfagas, predice los bloques muertos basándose en la secuencia de ráfagas que ha recibido un bloque.

2.1.2. Políticas de reemplazamiento híbridas

Generalmente, hay políticas de reemplazamiento que permiten obtener un buen rendimiento de los recursos de cache para un tipo de carga de trabajo, pero que su rendimiento decae si el comportamiento de la carga de trabajo cambia. En estos casos, lo más eficiente es implementar distintas políticas que se adapten dinámicamente a las diferentes cargas de trabajo.

En [QLMP06], se propone una política denominada *Sampling Based Adaptive Replacement* (SBAR), que se basa en la observación de que cuando ocurren varios fallos de cache al mismo tiempo, los ciclos de espera se amortizan al servir simultáneamente las peticiones a memoria. Para esto utiliza dos políticas: LRU y *Linear Policy* (LIN), donde esta última intenta mantener en cache los bloques que provocan fallos aislados.

Para la implementamentación de las dos políticas de reemplazamiento, utilizan un mecanismo al que denominan *Tournament Selection of Replacement Policy* (TSEL), el cual agrega al directorio principal de etiquetas dos direc-

torios auxiliares, de modo que cada uno de ellos implementa sólo una de las políticas y se utilizan para obtener estadísticas del comportamiento de la política que implementa. Para definir cuál de las políticas es mejor, se tiene un contador saturado que se incrementa cuando una de las políticas genera un fallo, y se decrementa cuando es la otra quien lo provoca. El incremento o decremento del contador, se realiza en un valor equivalente al costo del fallo provocado. De este modo, se decide la política más adecuada mediante el valor del bit más significativo del contador.

Una forma de implementar TSEL es agregar los directorios de etiquetas auxiliares para cada uno de los conjuntos de la cache, lo cual sería muy costoso, como se mencionó anteriormente. Para reducir el coste, se basan en que no es necesario que todos los conjuntos de la cache participen en la selección de la política, sino que es suficiente con muestrear el comportamiento de algunos conjuntos y extrapolar la decisión a los demás, seleccionando así la mejor política con una probabilidad muy alta. Según sus resultados, con 32 conjuntos se escogería la mejor política con una probabilidad mayor al 95 %. Los conjuntos que participan en la actualización se denominan *conjuntos líderes*, y los demás *conjuntos seguidores*. De este modo, sólo es necesario implementar los directorios de etiquetas auxiliares en los conjuntos líderes, lo que reduce de manera considerable el sobrecosto hardware. A este método lo denominan *Dynamic Set Sampling* (DSS) y ha sido muy empleado en trabajos posteriores.

En [QJP+07], se propone una mejora a esta técnica denominada política de inserción dinámica con *set dueling* (DIP-SD). En ella se elimina la necesidad de directorios auxiliares de etiquetas dedicando algunos conjuntos de la cache, entre 32 y 64 según su modelo teórico, a cada una de las políticas de

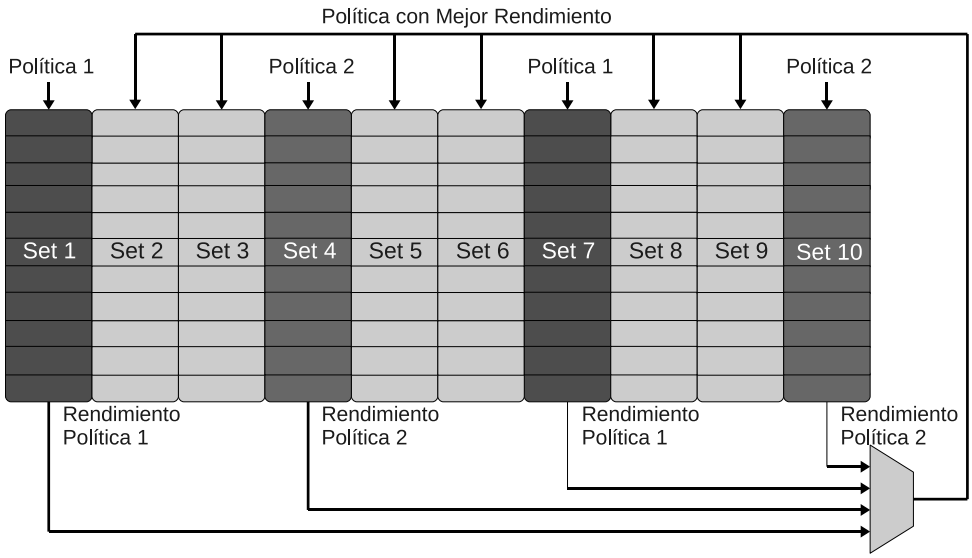

FIGURA 2.1: Esquema de cache con mecanismo de *set dueling*.

reemplazamiento (Figura 2.1). En este caso, las políticas utilizadas son LRU y *Bimodal Insertion Policy* (BIP). Esta última, inserta la mayoría de los bloques en la posición LRU y, con una probabilidad muy baja, los inserta en la posición MRU, logrando de este modo que algunos bloques apenas permanezcan en la cache y, por tanto, que no se polucione la cache con bloques que no serán reutilizados a corto plazo.

El desempeño de cada una de las políticas se continúa evaluando mediante un contador saturado que se incrementa con los fallos de una de las políticas y se decrementa con los fallos de la otra, de modo que, la política que produzca la menor cantidad de fallos en sus conjuntos dedicados es utilizada en los conjuntos seguidores.

2.2. Políticas *multicore*

En esta sección se presentan algunas de las propuestas concebidas para entornos multiprocesador/*multicore* que comparten un nivel de cache, las cuales consideran que en un momento determinado pueden estar en ejecución varios procesos simultáneamente y, por tanto, estar accediendo al mismo tiempo a memoria, lo que puede producir interferencia entre ellos. Las políticas presentadas intentan repartir los recursos de una manera eficiente para reducir así los efectos de la interferencia.

2.2.1. Particionado de la cache basado en la utilidad

En este caso, Qureshi y Patt [QP06] proponen realizar un particionamiento de la cache basándose en la utilidad que tendrá la partición de cache asignada a cada aplicación. Para esto, clasifican las aplicaciones en tres categorías dependiendo de cuánto se benefician con el incremento de la partición asignada. La primera categoría de aplicaciones no se beneficia significativamente a medida que la partición aumenta, por lo que se denominan de *baja utilidad*. La segunda categoría se beneficia a medida que se incrementa el tamaño de la partición, por lo que son llamadas de *utilidad alta*. La tercera categoría, son las aplicaciones que se benefician del incremento de tamaño de la partición sólo hasta cierto punto, por lo que son clasificadas como de *utilidad saturable*.

Para medir la utilidad de la cache, a cada *core* se le asigna un circuito de monitoreo de utilidad (*Utility Monitor* - UMON), que rastrea la información de la aplicación que se ejecuta en él. Para su implementación, se asigna un contador a cada vía de un conjunto, haciendo que este contador se incremente

cuando ocurra un acierto en dicha posición de la pila. Estos contadores representan el número de fallos evitados en cada posición de la pila. Además, se utiliza la técnica *Dynamic Set Sampling* (DSS) para reducir el número de conjuntos que dispondrán de estos contadores.

Luego, un algoritmo de particionamiento se ocupa de asignar el número de vías de la cache que pondrá a disposición de cada *core*, basándose en la información de los UMON. Para esto, intenta reducir el número de fallos total de todas las aplicaciones en ejecución. Este algoritmo se aplica una vez cada cinco millones de ciclos.

2.2.2. Políticas adaptativas de inserción para el manejo de caches compartidas

En esta propuesta [JHQ+08], se basan en la idea de asignar más recursos de cache a las aplicaciones que se benefician de ella y menos recursos a las aplicaciones que no se benefician. Para esto, al igual que en el trabajo descrito en la sección anterior, definen una clasificación de las aplicaciones basándose en el beneficio que obtienen de la cache. Si se considera una memoria cache de tamaño N y que aplica la política LRU, las aplicaciones se clasifican en:

- Aplicaciones amistosas con la cache: Estas aplicaciones se benefician en relación lineal a la cantidad de cache que se les asigna. En este caso, la política LRU funciona adecuadamente.

- Aplicaciones que caben en la cache: Estas aplicaciones tienen un conjunto de trabajo de tamaño M, con $M < N$, por lo que si se les asigna un

tamaño de cache menor que M, empiezan a funcionar mal. En general, funcionan correctamente usando la política LRU.

- Aplicaciones que causan *thrashing*: Estas aplicaciones tienen un conjunto de trabajo de un tamaño M, con $M > N$, por lo que presentan poco reuso de los datos. Estas aplicaciones no funcionan bien usando la política LRU.

- Aplicaciones de tipo *streaming*: Estas aplicaciones tienen un conjunto de trabajo extremadamente grande y poco reuso de la cache, por lo que causan *thrashing* al usar cualquier política de reemplazamiento.

Lo que se pretende en este caso, es decidir entre dos políticas de reemplazamiento de la cache, LRU y BIP. Así, a las aplicaciones de las dos primeras categorias se les asigna una mayor cantidad de recursos de cache, pues sacarían provecho de ellos, por lo que se utiliza la política LRU. Por el contrario, a las aplicaciones de la tercera y cuarta categoría, se les asignan pocos recursos de cache, ya que no se benefician de ellos, por tanto se utiliza BIP para eliminar sus bloques más rápidamente.

Esta decisión se toma para cada una de las aplicaciones que se ejecuten de manera concurrente. A esta técnica se le denomina *Thread-Aware Dynamic Insertion Policy* (TADIP) y se plantean dos implementaciones de la misma.

La primera implementación se denomina TADIP-*Isolated* debido a que, para escoger la mejor política, asume que cada una de las aplicaciones se está ejecutando aislada de las otras. La segunda implementación, llamada TADIP-*Feedback*, intenta tomar en consideración el efecto de la política de reemplazamiento utilizada en una aplicación sobre las demás.

2.3. La política *Probabilistic Escape Lifo*

Esta propuesta, realizada por Chaudhuri en [Cha09b] y complementada en [Cha09a], recibirá especial atención, pues en ella se define la política de reemplazamiento *probabilistic escape lifo* (peLifo), en la cual se basa este trabajo.

La idea principal en peLifo es que los bloques de la cache generalmente muestran pocos usos durante su tiempo de vida en la cache de último nivel, además, en promedio, este número es menor que la asociatividad de la cache. Esta propiedad se ha aprovechado anteriormente mediante técnicas de detección de bloques muertos y de políticas de inserción dinámica como las presentadas en la sección 2.1.

Sin embargo, con el aumento de capacidad y de asociatividad de la cache de último nivel, las políticas de reemplazamiento basadas en predictores de bloques muertos, tienden a comportarse como la política LRU, puesto que en general, el bloque LRU está muerto, ya que no tendrá más reusos a corto plazo. Por otro lado, las políticas de inserción dinámica presentan poco beneficio, pues tienden a reemplazar prematuramente bloques con una cantidad de usos pequeña, pero mayor que uno.

PeLifo se sitúa en un punto medio entre las dos técnicas anteriores, pues busca realizar los reemplazamientos entre los bloques que han ingresado recientemente (siempre que ya hayan satisfecho sus aciertos a corto plazo), lo que permite reservar las demás posiciones, donde estarán los bloques que han ingresado hace más tiempo, para reusos a largo plazo.

Buscando este objetivo, peLifo realiza una implementación mejorada de la

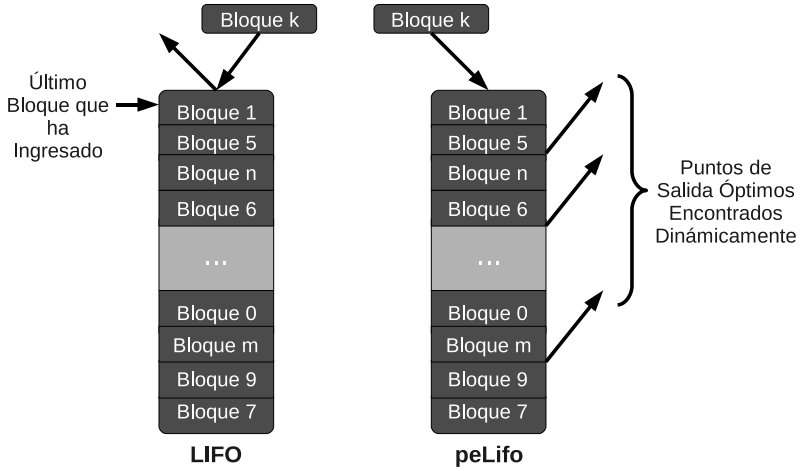

FIGURA 2.2: Reemplazamiento en las políticas LIFO y peLifo.

política *last in first out* (LIFO), la cual reemplaza siempre el último bloque que ha ingresado en el conjunto. Para esto, define el *fill stack*, o pila de llenado, en el que la primera posición contendrá el último bloque que ha sido ingresado en el conjunto y en la última posición estará el bloque menos recientemente ingresado, es decir, los bloques siempre ingresarán por la parte alta del *fill stack*. A diferencia de LIFO, peLifo reemplaza bloques en puntos del *fill stack* encontrados dinámicamente, a partir de los cuales existe menor probabilidad de que un bloque tenga más reusos a corto plazo (Figura 2.2).

2.3.1. Cálculo de los puntos de escape

Para obtener estos puntos, se define la probabilidad de escape en una posición k del *fill stack* como la probabilidad de que los bloques de la cache que están en posiciones del *fill stack* mayores que k generen aciertos, y se puede calcular como el número de bloques de cache que generan al menos un

acierto en posiciones del *fill stack* mayores a k dividido por el número total de bloques ingresados en la cache. Esta probabilidad se calcula periódicamente cada N ingresos.

Para el cálculo de esta probabilidad, se utiliza un *array* de A contadores saturados, denominado *epCounter*, donde A es la asociatividad de la cache, y un *array* donde se mantiene la posición del *fill stack* donde cada bloque tuvo su último acierto. El *array epCounter* se actualiza de la siguiente manera: si el bloque genera un acierto en la posición k del *fill stack*, todas las posiciones del *epCounter* desde la posición del último acierto hasta $k-1$ se incrementan y se actualiza la última posición de acierto del bloque a k. Es de notar, que cuando ingresa un bloque, no se incrementa ningún contador, pero la posición de último acierto se inicializa a cero. Una vez se tienen estos datos, se les aplica una serie de aproximaciones para facilitar su manejo en hardware, las cuales se enuncian a continuación:

- Primero, cada valor del *epCounter* diferente de cero se redondea a la siguiente potencia de dos, si no lo es ya.
- Luego, cada valor se reemplaza por el logaritmo en base dos, dejando sin cambios los valores iguales a cero.
- Por último, se substraen estos valores al logaritmo en base dos de N y se almacenan de nuevo en el *array epCounter*. Además, en este punto se realiza una copia de *epCounter* a otro *array* denominado *epCounter-LastEpoch*, que será utilizado posteriormente.

El siguiente paso es identificar los puntos donde la probabilidad de escape tiene un descenso significativo. Estos puntos se denominan *puntos de escape* y

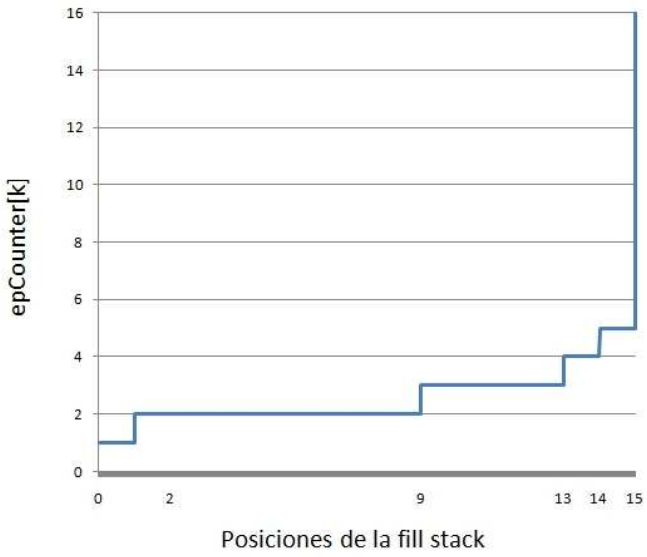

FIGURA 2.3: *epCounter* respecto a la posición k del *fill stack*.

se corresponden con los puntos de salida óptimos de la cache (Figura 2.2), y se ha observado que con tres de ellos se logra captar completamente la dinámica de la parte superior del *fill stack*. En la figura 2.3, se puede observar como el valor de *epCounter* crece monótonamente y, por tanto, decrece la probabilidad de escape. Se pueden ver, además, varios puntos donde el valor de *epCounter* se incrementa, estos puntos corresponden a los puntos de escape. Cabe resaltar que existirá un punto de escape en la posición cero si el valor de *epCounter* en esta posición es mayor que cero. Para el cálculo de los demás puntos de escape se introduce el concepto de *cluster*, como el conjunto de posiciones consecutivas del *fill stack* que tienen un valor constante de la probabilidad de escape, de modo que se definirá un punto de escape en el centro de cada *cluster*, el cual viene dado por $\lfloor (posIni + posFin + 1)/2 \rfloor$.

Existen dos casos que requieren tratamiento especial. El primero, es cuando

las posiciones iniciales de *epCounter* son cero, entonces el primer punto de escape no será en el centro del *cluster*, sino en el primer cambio detectado en la probabilidad de escape. Esto se debe a que, dada la definición de la probabilidad de escape, los bloques en el centro de este *cluster* continuarán generando aciertos hasta llegar al primer cambio. El segundo caso, aparece cuando dos puntos de escape son iguales y se resuelve incrementando en uno el último punto de escape que fue calculado.

2.3.2. Determinación dinámica del punto de escape óptimo y aplicación de la política

Como se mencionó anteriormente, con tres puntos de escape se puede captar la dinámica del *fill stack*, por lo que se definen cuatro políticas a escoger, P_1 a P_4, siendo P_4 la política LRU tradicional. La política P_i, con $i \in \{1, 2, 3\}$, corresponde al proceso ejecutado cuando se ha escogido el punto de escape i. Así, para aprender dinámicamente el punto de escape más apropiado entre los tres que se calculan, se utiliza la técnica de *set dueling*, descrita en la sección 2.1.2.

Para implementar la técnica de *set dueling*, dentro de cada par de bancos de la cache, se dedica un pequeño número de conjuntos a cada política, de modo que, cada par de bancos pueda implementar independientemente la política más adecuada.

Para el cálculo dinámico de la política más apropiada, se mantienen seis contadores saturados, cada uno inicializado al punto medio, M, de su rango. Estos contadores se denotan por C_{ij}, con $i<j$ y $i, j \in \{1, 2, 3, 4\}$. De modo

que, un fallo en un conjunto dedicado a la política P_i, incrementa todos los contadores C_{ij} y decrementa todos los contadores C_{ji}. Así, para $i<j$, la política P_j es mejor que la política P_i si $C_{ij} \leq M$; de lo contrario, la política P_i es mejor. Por lo tanto, si $i>j$, la política P_j es mejor que la política P_i si $C_{ji}<M$. Una política en particular, será la ganadora si los tres contadores en los que participa lo indican.

Una vez se tiene seleccionado el punto de escape a utilizar, p_i, se reemplazará el bloque más cercano a la cabeza del *fill stack*, que satisfaga los siguientes criterios:

- El bloque no ha generado un acierto en su posición actual en el *fill stack*.
- Su posición actual en el *fill stack* es mayor o igual que el punto de escape p_i.

Si no se encuentra ningún bloque que satisfaga las anteriores condiciones, se aplica la política LRU.

2.3.3. Reevaluación periódica de los puntos de escape

A lo largo de su ejecución, una aplicación pasa por diferentes fases con distintas características, por lo que los puntos de escape varían y deben ser recalculados.

Así, cuando la cache empieza a funcionar (Figura 2.4), lo hace en el modo LRU, en el cual se aplica la política LRU en todos los conjuntos. Este modo tiene una duración de $N = 2^n$ ingresos, y se denomina época LRU. Los puntos de escape siempre serán calculados al terminar una época LRU, para luego

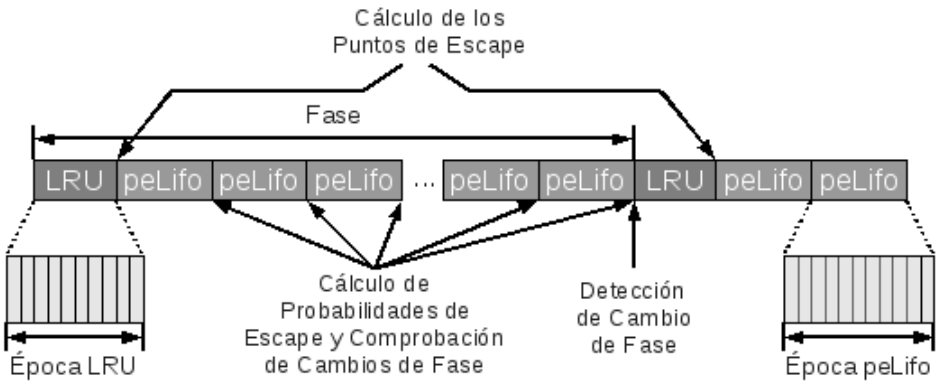

FIGURA 2.4: Esquema de funcionamiento de la política peLifo

cambiar a modo de funcionamiento peLifo, en el cual, el *array epCounter* se actualiza sólo con los datos de los conjuntos dedicados a la política LRU. Debido a que los conjuntos dedicados a evaluar cada política son pocos, una época peLifo tendrá una duración de $N' = 2^{n'} < 2^n$.

Al finalizar cada época peLifo se debe determinar si ha ocurrido un cambio de fase en la aplicación. Para esto, se comparan las probabilidades de escape de la época actual y de la época anterior, de modo que si se detecta una diferencia de al menos D en alguna de las posiciones del *array* por encima del tercer punto de escape se declara un cambio de fase.

Al detectar un cambio de fase, se pasa al modo LRU y se recalculan los puntos de escape siguiendo el procedimiento descrito anteriormente. La razón de hacer que todos los conjuntos utilicen la política LRU es la de "limpiar" la parte baja del *fill stack*, de modo que ésta se rellene con bloques de la nueva fase de ejecución.

2.3.4. Hardware extra necesario

Por último, el hardware adicional necesario para la implementación de esta política es el siguiente (siendo A la asociatividad de la cache):

- $Alog_2(A)$ bits por conjunto para almacenar la posición de cada bloque en la *fill stack*. Estos bits son adicionales a los que requiere la *recency stack*, para la implementación de la política LRU.
- Dos *arrays* de A posiciones para *epCounter* y *epCounterLastEpoch*.
- Espacio para almacenar los tres puntos de escape.
- Seis contadores saturados para la elección de política.
- $Alog_2(A)$ bits por conjunto, para almacenar la posición del último acierto recibido en cada bloque.
- Un bit por bloque para determinar si ha producido un acierto en su posición actual en el *fill stack*.

En total, el hardware necesario en la implementación puede ser desde un quinto hasta la mitad del necesario para la implementación de una política de reemplazamiento actual basada en predicción de bloques muertos.

Capítulo 3

Trabajo realizado

Como se mencionó anteriormente, el objetivo de este trabajo es realizar una extensión a entornos *multicore* de la política peLifo original. Para esto, se proponen algunas modificaciones a dicha política, de modo que se tenga disponible información sobre las características de los procesos que se ejecutan en cada momento.

Para realizar esta extensión es necesario, en primer lugar, poder distinguir el comportamiento de cada una de las aplicaciones en ejecución, pero en la implementación básica de la cache esto no es posible, ya que no se guarda información concerniente a los procesos que realizan las peticiones a memoria que ocasionan que los bloques sean almacenados en la cache. Para solucionar esta situación, se agrega un campo a cada bloque donde se almacena un identificador que indica el proceso que realizó la petición que lo ingresó en memoria (Figura 3.1).

Las modificaciones propuestas para la política peLifo se describen en las siguientes secciones.

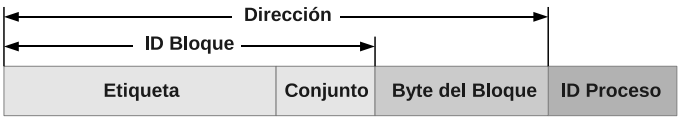

FIGURA 3.1: Organización lógica de la cache modificada

3.1. PeLifo por listas

La finalidad de esta modificación es asignar más recursos a aquellos procesos que mejor están utilizando la cache. Es importante recordar que peLifo reserva las partes bajas del *fill stack* para bloques que pueden generar aciertos a largo plazo. Es posible que los bloques que entran en esa zona (algo que no controla la política) pertenezcan a procesos cuyos bloques no presentarán más aciertos, y que, por tanto, estén ocupando espacio en el conjunto de la cache que podría ser utilizado por un bloque de un proceso con más utilidad. Se propone medir dinámicamente la proporción de aciertos por bloque en las zonas bajas del *fill stack* y establecer una lista ordenada de todos los procesos, de modo que el proceso que tenga la menor proporción de aciertos por bloque, sea el primero en la lista, y el que tenga la mayor proporción sea último. A continuación, los bloques serán reemplazados de la cache según el orden de la lista.

Para realizar la medida de la proporción, se lleva la cuenta de la cantidad de bloques que tiene cada proceso en la cache y la cantidad de aciertos producidos. Estos conteos se realizan a partir de cada punto de escape, de modo que, al final de la época, se obtiene el número de aciertos por bloque, para cada proceso t_i y a partir de cada punto de escape p_i, usando la fórmula:

$$proporcion(t_i, p_i) = \frac{aciertos(t_i, p_i)}{bloques(t_i, p_i)} \tag{3.1}$$

Una vez se tiene este cálculo, se realiza la lista ordenada de los procesos para cada punto de escape, obteniéndose así tres listas. De modo que, si se emplea el punto de escape p_i, se reemplazará el bloque más cercano a la cabeza del *fill stack* que satisfaga los siguientes tres criterios:

- No ha producido ningún acierto en su posición actual en el *fill stack*.
- Su posición actual en el *fill stack* es mayor o igual que el punto de escape correspondiente.
- El bloque pertenece al proceso con menor proporción de aciertos por bloque de la lista.

Si no se encuentra un bloque con estas condiciones, se vuelve a la primera condición, pero usando el siguiente proceso de la lista. Finalmente, en caso de no encontrar ningún bloque, se aplica la política LRU.

3.1.1. Simplificaciones

Dado que el sistema propuesto debe ser implementado en hardware, se han planteado dos simplificaciones para reducir su sobrecosto.

- La primera simplificación que se propone es, en lugar de llevar la cuenta de la cantidad de bloques a partir de cada punto de escape, lo cual es muy costoso, se mantiene sólo el total de bloques en toda la cache que tiene cada uno de los procesos en ejecución. Con esto se reduce la

cantidad de contadores necesarios para obtener el total de bloques a uno por proceso.

- La segunda simplificación se basa en que la proporción de aciertos por bloque no necesita ser muy precisa, pues ésta solo es utilizada para generar la lista ordenada. Teniendo esto en cuenta, es posible simplificar la división necesaria en el cálculo de la proporción aproximando el número de bloques total a la siguiente potencia de dos, de modo que la división se convierta en un desplazamiento a la derecha igual a la potencia de dos a la que fue aproximado el número de bloques del proceso.

Con estas simplificaciones, el hardware adicional necesario para implementar esta propuesta para un entorno de cuatro *cores*, se muestra en la tabla 3.1:

Hardware	**Cantidad**	**Uso**	**Tamaño**
Contadores	4	Conteo de número de bloques en la cache por proceso	16 bits
Contadores	12	Conteo de número de aciertos a partir de cada punto de escape para cada proceso	25 bits
Reg. Desplazamiento	1	Cálculo de Divisiones	32 bits

TABLA 3.1: Hardware adicional necesario para la implementación de la extensión por listas para un entorno de cuatro *cores*.

3.2. PeLifo proporcional

El propósito de esta modificación es el mismo que en el caso anterior: extraer los bloques de procesos que no sacan partido a los aciertos a largo plazo y conservar aquellos que si lo hacen, pero, en lugar de utilizar una lista de prioridades, se calcula una proporción basada en el uso que le está dando

cada proceso a la cache, y se reemplazarán bloques de cada proceso basándose en dicha proporción.

La métrica utilizada en este caso, es el inverso de la proporción utilizada en el caso anterior, la cual, para un proceso t_i y para un punto de escape p_i, se calcula utilizando la fórmula:

$$proporcionInversa(t_i, p_i) = \frac{bloques(t_i, p_i)}{aciertos(t_i, p_i)} \tag{3.2}$$

La razón de invertir la métrica es que en este caso se requiere que el menor valor corresponda al proceso que mejor rendimiento por bloque está obteniendo, contrario a lo que se buscaba en el caso anterior.

Una vez se tienen las proporciones para cada proceso a partir de cada punto de escape, se obtiene el total de todas las proporciones correspondientes a cada punto de escape, así:

$$proporcionTotal(p_i) = \sum_{t_i} proporcionInversa(t_i, p_i) \tag{3.3}$$

Por último, se encuentra la proporción respecto al total de cada una de las proporciones correspondientes a cada punto de escape y se escala este valor con un número que corresponde a la cantidad total de bloques que se quieren reemplazar de todos los procesos, antes de reiniciar el conteo de bloques reemplazados. Para esto se aplica la fórmula:

$$numBloques(t_i, p_i) = factorEscala * \frac{proporcionInversa(t_i, p_i)}{proporcionTotal(p_i)} \tag{3.4}$$

Una vez se tienen las proporciones calculadas por proceso y por punto de escape, se inicializa un contador con dicho valor para cada proceso en cada punto de escape. De modo que, si se va a utilizar el punto de escape p_i, se reemplazará el bloque más cercano a la cabeza del *fill stack* que satisfaga las siguientes condiciones:

- No ha producido ningún acierto en su posición actual en el *fill stack*.
- Su posición actual en el *fill stack* es mayor o igual que el punto de escape correspondiente.
- El contador del proceso que ha ingresado el bloque es mayor que cero.

Si algún bloque del conjunto cumple todas las condiciones, se decrementa el contador del proceso correspondiente, de modo que cuando todos los contadores asociados a un punto de escape lleguen a cero se reinician a sus valores calculados para la época actual. Si no se encuentra ningún bloque que cumpla las condiciones, se aplica la política peLifo original.

3.2.1. Simplificaciones

En este caso se proponen tres simplificaciones para que sea viable la construcción de este sistema en hardware:

- La primera simplificación es la misma que en la propuesta anterior, es decir, en lugar de tener un contador para los bloques de cada proceso a partir de cada punto de escape se tendría un contador por cada proceso, en el cual se almacena la cantidad total de bloques que tiene en la cache.

- La segunda simplificación es para facilitar el cálculo de la proporción inversa (fórmula 3.2). La idea es eliminar la necesidad de realizar una división en el cálculo. Para esto, se escala el valor del número de bloques del proceso, ya que en general es mas pequeño que la cantidad de aciertos, por un número que sea potencia de dos, en este trabajo se utiliza 256. Luego, se aproxima el número de aciertos a la siguiente potencia de dos, de modo que la división se convierta en un desplazamiento a la derecha igual a la potencia de dos a la que fue aproximado el número de aciertos.

- La última simplificación se aplica al calcular el número de bloques a reemplazar de cada proceso (fórmula 3.4). La primera consideración es para convertir la multiplicación en un desplazamiento a la izquierda. Para esto, se multiplica por un factor de escala que sea potencia de dos. En la simulación realizada se utilizó el número 128. Además, en lugar de calcular la división, se aproxima la proporción total de cada punto de escape a la siguiente potencia de dos, para así convertirla en un desplazamiento a la derecha igual a la potencia encontrada.

Estas simplificaciones funcionan debido a que el valor que se busca es una proporción, por lo que al ejecutarlas, dicha proporción se mantiene aproximadamente, como se puede comprobar con los resultados obtenidos en el capítulo 5. De este modo, en la tabla 3.2 se muestra el hardware adicional necesario para la implementación en un entorno de cuatro *cores*.

Hardware	Cantidad	Uso	Tamaño
Contadores	4	Conteo de número de bloques en la cache por proceso	16 bits
Contadores	12	Conteo de número de aciertos a partir de cada punto de escape para cada proceso	25 bits
Reg. Desplazamiento	1	Cálculo de multiplicaciones	32 bits
Reg. Desplazamiento	1	Cálculo de divisiones	32 bits
Sumador	1	Cálculo de proporción total	32 bits
Registros	12	Cantidad de bloques a reemplazar por cada proceso a partir de cada punto de escape	8 bits
Compuertas	12	Detección de valor cero	8 bits

TABLA 3.2: Hardware necesario para la implementación de la extensión proporcional en un entorno de cuatro *cores*.

3.3. PeLifo proporcional - por listas

En los resultados preliminares, se observó que la política peLifo por listas funciona bien en algunos casos en los que la política peLifo proporcional funciona un poco peor y viceversa. A causa de esto, se realizó una implementación combinando estas dos políticas para intentar obtener mejores resultados. Además, dado que las métricas utilizadas en ambas políticas se calculan a partir de los mismos datos, no se incurre en un sobrecosto hardware muy elevado al realizar una implementación conjunta de estas dos políticas.

Así, cuando se va a reemplazar un bloque, se aplica, en primera instancia, la política peLifo proporcional y, en caso de que no se encuentre ningún bloque que cumpla las condiciones anteriormente establecidas para esta política, se aplica la política peLifo por listas, en lugar de la política peLifo original.

Capítulo 4

Entorno experimental

En esta sección se hace una descripción del simulador utilizado en el desarrollo de este trabajo, así como de las herramientas que fueron utilizadas para su implementación.

En términos generales, el entorno de simulación utiliza la herramienta de instrumentación dinámica Pin para obtener las trazas de acceso a memoria generadas por las aplicaciones. Luego estas trazas son pasadas al simulador, que se basa en el modelo de la memoria cache implementada por el simulador arquitectónico SESC, para evaluar su rendimiento.

4.1. Pin

La instrumentación es una técnica que consiste en agregar nuevas instrucciones a una aplicación, generalmente con el fin de tomar medidas de su rendimiento. Además, es posible obtener información que sólo está disponible en tiempo de ejecución, como las direcciones de memoria que utiliza la apli-

cación, el contenido de los registros del procesador, etc. Existen dos tipos de instrumentación, estática y dinámica. La instrumentación estática se realiza agregando directamente las nuevas instrucciones en el código de la aplicación, lo que hace que tenga bastantes limitaciones. En contraposición, la instrumentación dinámica se realiza en tiempo de ejecución, lo que le permite tener mayor versatilidad, además de no necesitar el código fuente de la aplicación que se quiere instrumentar. Una de las desventajas más importantes de la instrumentación es que, dado que se insertan nuevas instrucciones en la aplicación, se produce un incremento en el tiempo de ejecución que puede llegar a ser muy significativo.

Pin [pina] es una herramienta, desarrollada por *Intel Corporation* y la Universidad de Colorado, utilizada para la instrumentación dinámica de binarios que permite inyectar código arbitrario, escrito en C o C++, en cualquier lugar del ejecutable. Para esto, es necesario definir dos tipos de rutinas: rutinas de instrumentación y rutinas de análisis. Las rutinas de instrumentación definen en qué puntos del código se van a agregar las nuevas instrucciones y las rutinas de análisis determinan las instrucciones que serán insertadas en el código.

Al nivel más alto, Pin está compuesto por una máquina virtual (*Virtual Machine* - VM), una cache de código y una API (*Application Programming Interface*) de instrumentación. La VM contiene un compilador *just-in-time* (JIT), un emulador y un despachador [LCM+05]. Una vez que Pin toma control de la aplicación, la VM coordina sus componentes para ejecutarla. El JIT compila e instrumenta el código de la aplicación, el cual es lanzado luego por el despachador. El código compilado es almacenado en la cache de código, para su posterior uso. El emulador interpreta las instrucciones que se pueden

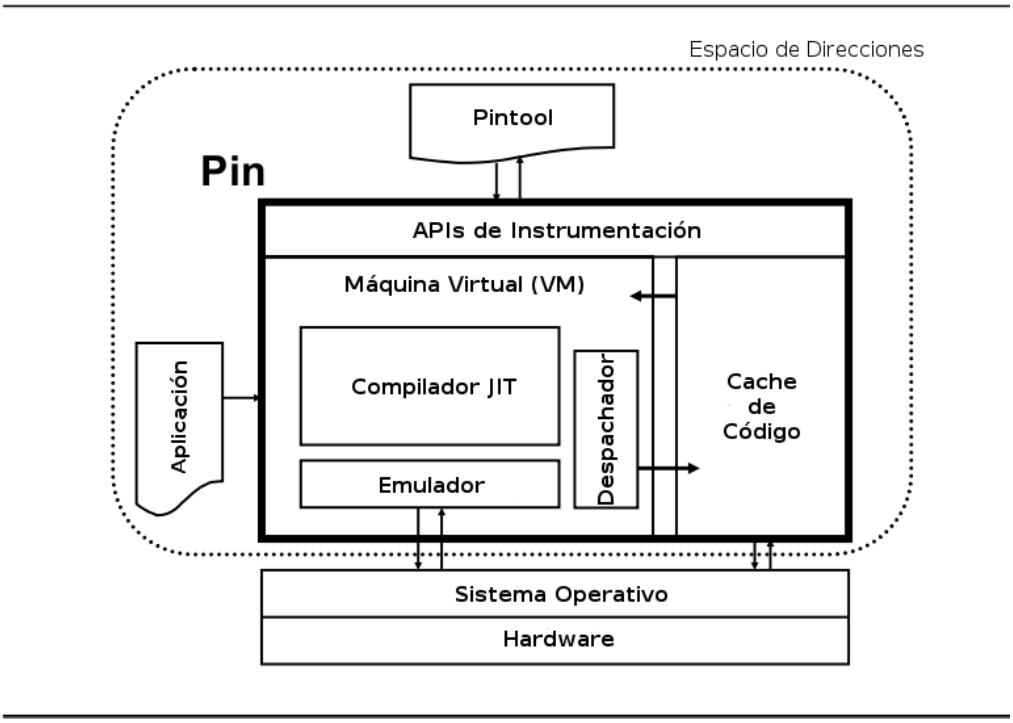

FIGURA 4.1: Estructura del software Pin [LCM+05].

ejecutar directamente, lo que se usa especialmente con llamadas a sistema que requieren tratamiento especial por parte de la VM. Dado que Pin está por encima del sistema operativo, sólo puede capturar código de nivel de usuario.

Como se muestra en la figura 4.1, hay tres programas en ejecución cuando se ejecuta una aplicación instrumentada: la aplicación, Pin y la Pintool. Pin es el motor que compila e instrumenta la aplicación. La Pintool contiene las rutinas de instrumentación y análisis y es enlazada con una librería que le permite comunicarse con Pin.

Para la inyección de código, se carga Pin en el espacio de direcciones de la aplicación, luego se hace uso de la API *Ptrace* para obtener el control de la aplicación y capturar el contexto del procesador. Para ésto, carga el binario

de Pin en el espacio de direcciones de la aplicación y lo ejecuta, el cual, por su parte, se inicializa para luego cargar la Pintool en el mismo espacio de direcciones y ejecutarla. La Pintool se inicializa y luego solicita a Pin que ejecute la aplicación. Pin crea un contexto inicial y le pasa el código de la aplicación al compilador JIT, ya sea a partir del punto de entrada, o a partir del PC actual si se ha pegado (*attach*) a la aplicación. Usar Ptrace como el método de inyección de código, permite pegarse (*attach*) a una aplicación que se encuentra ya en ejecución de la misma forma que lo hace un depurador. También es posible despegarse (*detach*) de un proceso y continuar su ejecución sin instrumentación.

Se decidió utilizar Pin para la simulación arquitectónica por las siguientes razones [MF05]:

- Facilidad de implementación, pues permite acceder a cada una de las instrucciones del flujo de instrucciones, y provee métodos para distinguir sus características más importantes, lo que permite evitar todo el trabajo de interpretación y decodificación de las instrucciones.

- Rendimiento, ya que las instrucciones, luego de una instrumentación inicial realizada por Pin, se ejecutan directamente en el hardware, en lugar de a través de un intérprete.

- Simulación multitarea, pues, agregando el uso de memoria compartida, se pueden evaluar las consecuencias de la interacción de diversas aplicaciones en una ejecución real.

4.2. SESC

SESC [ses] es un simulador arquitectónico desarrollado principalmente por el grupo de investigación i-acoma de la universidad UIUC y varios grupos de otras universidades que modelan diferentes arquitecturas tales como procesadores simples, CMPs, entre otros. SESC modela un *pipeline* fuera de orden completo con predicción de saltos, caches, buses, y cualquier otro componente de un procesador actual necesario para una simulación precisa.

En SESC las instrucciones se ejecutan en un módulo de emulación, el cual emula el conjunto de instrucciones de la arquitectura MIPS. Este módulo emula las instrucciones en el orden binario de la aplicación y está construido a partir de MINT, un emulador de un procesador MIPS.

El emulador le entrega objetos de instrucciones (*instruction objects*) a SESC que luego son usados para la simulación temporal. Dichos objetos de instrucciones contienen toda la información relevante necesaria para una temporización precisa. Éstos incluyen la dirección de la instrucción, la dirección de cualquier lectura o escritura a memoria, los registros destino y fuente, y las unidades funcionales usadas por la instrucción. El resto del simulador utiliza esta información para calcular cuanto tiempo le lleva a la instrucción ejecutarse a través del *pipeline*.

Hay dos razones para esta implementación. La primera es que es mucho más rápido tener la instrucción actual ejecutada en un emulador simple. La segunda, es que es más fácil programar y depurar cuando la ejecución y la temporización están separadas. Es posible que el simulador temporal no tenga uns precisión del 100 %, siempre y cuando, esto no afecte la ejecución de las

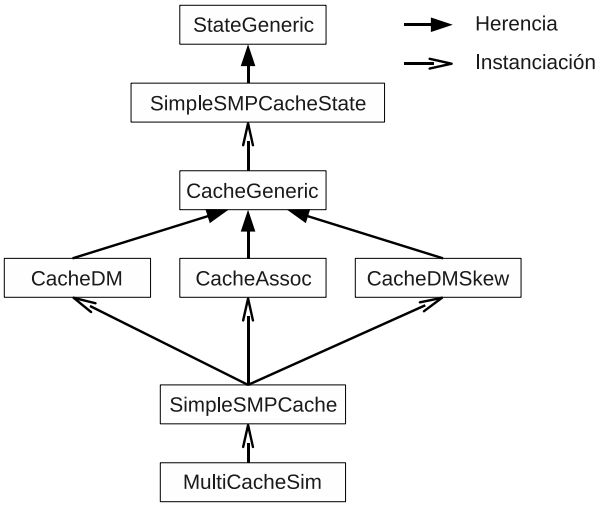

FIGURA 4.2: Jerarquía de clases del simulador básico

instrucciones.

4.3. Simulador

4.3.1. Simulador básico

Ya que SESC simula un sistema completo y, para este caso, no era de interés simular el sistema en su totalidad, se ha utilizado como base para el desarrollo del proyecto el simulador *MultiCacheSim*, el cual permite simular caches en un entorno multiprocesador, está escrito en C++ y está basado en la cache implementada en SESC. Su jerarquía de clases simplificada se muestra en la figura 4.2. *MultiCacheSim* fue desarrollado por Brandon Lucia en la universidad de Washington [mul].

Para la simulación del funcionamiento de la cache, utiliza la clase *CacheGeneric*, cuya implementación fue tomada del simulador SESC anteriormente

descrito. Esta clase se encarga de implementar el funcionamiento de la cache, como el cálculo de etiquetas, políticas de reemplazamiento (LRU y aleatoria), etc.

En el simulador SESC, el estado de coherencia de cada línea de la cache esta implementado mediante un objeto *StateGeneric*. En *MultiCacheSim* se ha definido la clase *simpleSMPCacheState*, que hereda de la clase de estado en SESC, y que implementa el protocolo de coherencia MESI.

Utiliza luego, la clase *SimpleSMPCache*, para encapsular las clases que definen la cache y su estado, además de proveer métodos de lectura y escritura. Por último, se utiliza la clase *MultiCacheSim*, que sirve de contenedor para todos las caches de primer nivel que se definan.

Para generar las trazas de acceso a memoria necesarias para el simulador, se utiliza la herramienta Pin. Se desarrolló una *Pintool* en la que se define un objeto de la clase *MultiCacheSim*, que simulará la jerarquía de cache, y cuya rutina de instrumentación identifica los accesos a memoria, e inserta llamadas a las funciones de lectura/escritura de la jerarquía de memoria definida. El esquema del simulador base se presenta en la figura 4.3.

4.3.2. Simulador modificado

MultiCacheSim presenta varias limitaciones importantes que fue necesario resolver para la realización de este trabajo. En esta sección se presentan las modificaciones que fueron efectuadas en el simulador básico.

El primero de los aspectos a tratar en el simulador, fue que la jerarquía de cache de interés es de varios niveles. Para solucionarlo se agregaron métodos

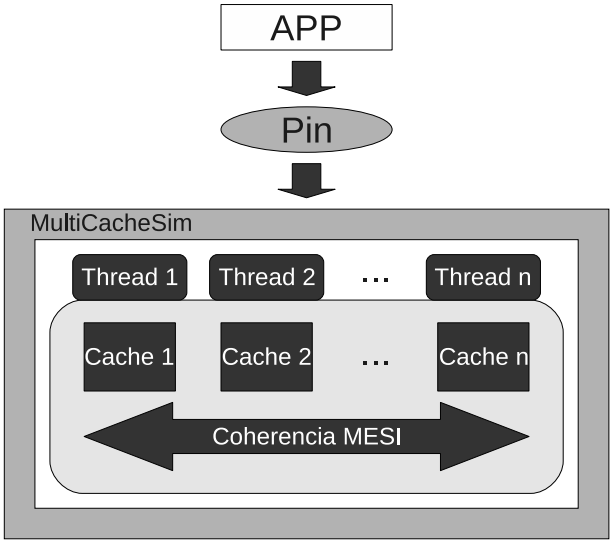

FIGURA 4.3: Esquema del simulador básico

que buscan inicialmente los datos en el primer nivel de la cache, y en caso de fallo, se verifica si existe un nivel superior, en cuyo caso se continúa la búsqueda en dicho nivel. Este proceso se repite hasta que no se encuentren más niveles superiores. De este modo, es posible definir una jerarquía de cache con cualquier número de niveles.

La política de escritura entre niveles de cache definida en la implementación fue *write back*, por ser la que tiene un uso más extendido. Además de esto, se determinó que los niveles de cache fueran inclusivos, y como se pretendía tener distintas políticas de reemplazamiento en cada nivel, fue necesario utilizar un mecanismo de invalidación de líneas desde los niveles superiores a los inferiores.

Para la implementación de la política de reemplazamiento peLifo en *MultiCacheSim*, se ha utilizado el código encontrado en [pel] desarrollado por el autor original de peLifo. Este código, implementa los métodos principales para la gestión de la política peLifo y para adaptarlo al simulador, se definió una

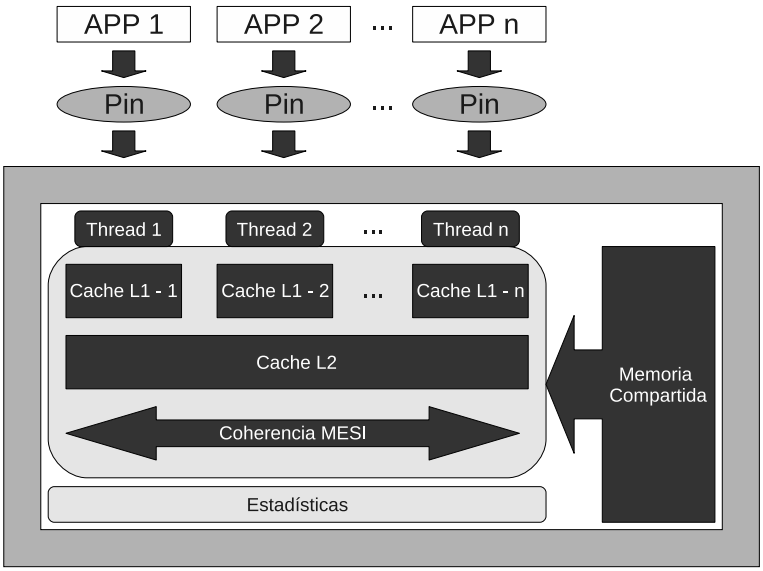

FIGURA 4.4: Esquema del simulador modificado.

nueva clase, *CachePeLifo*, que hereda de *CacheGeneric* e implementa la nueva política de reemplazamiento.

El inconveniente más significativo que tenía el simulador, es que, si bien era posible realizar medidas de aplicaciones *multithread*, no era posible realizar simulaciones que contuvieran información de distintas aplicaciones simultáneamente. Para abordar este problema, se optó por el uso de memoria compartida, de modo que se crea una instancia del simulador para cada proceso que participe en la simulación, y los datos que deben ser comunes entre ellos, tales como el estado de la cache y las medidas, se sitúan en un espacio de memoria compartida de modo que se pueda acceder a ellos desde cualquier instancia del simulador. El esquema utilizado se muestra en la figura 4.4. Algo similar fue realizado en [JCLJ08] con buenos resultados.

Para lograr esto se hizo uso de la librería del sistema <sys/shm.h>. En la

práctica, es complicado definir las diferentes instancias de los objetos directamente en memoria compartida, por lo que se optó por definirlos de manera privada en cada una de las instancias del simulador, y luego mapear los atributos, siempre que fueran tipos básicos o estructuras, en memoria compartida.

Para asegurar que las distintas instancias del simulador no modifiquen simultáneamente los segmentos de memoria compartida definidos, se utilizan mecanismos de exclusión mutua que hacen que sólo un proceso a la vez pueda acceder a dichos segmentos. Esto presenta un inconveniente, los accesos a memoria se serializan, lo que hace que el tiempo de simulación se incremente y además puede que no represente de una manera exacta lo que ocurre en un sistema hardware.

Por último, para el uso de los *simpoints* (descritos en la sección 4.3.3), se implementó una barrera de sincronización, de modo que cuando uno de los procesos llega a su *simpoint*, verifica si los demás ya llegaron, y si no es así, se detiene hasta que todos los procesos estén en dicho punto, logrando así que todos empiecen simultáneamente. Al finalizar mil millones de instrucciones, el proceso verifica si es el último en terminar, si lo es, termina su ejecución, de lo contrario continúa su ejecución, pero sin afectar a las estadísticas medidas. Además, después de cada instrucción ejecutada, verifica si ya han terminado todos, en cuyo caso finaliza su ejecución.

4.3.3. Simpoints

Se optó por no realizar las simulaciones con la totalidad de las instrucciones de las aplicaciones por las siguientes razones:

- Debido a que se hicieron simulaciones para entornos multitarea, no todas las aplicaciones tienen la misma duración y, por tanto, si se simularan en su totalidad, no se podría medir adecuadamente el comportamiento de la cache debido a su interacción.

- Al utilizar los conjuntos de referencia como entradas de las aplicaciones, la simulación completa de éstas tomaría demasiado tiempo, y los resultados que se podrían obtener en comparación con una ejecución parcial no compensan la inversión del tiempo adicional.

Para solucionar esto, se utilizó la herramienta *Simpoint* [sim] en su versión 3.2. Esta herramienta calcula fases para un par aplicación/entrada, y luego escoge un solo representante para cada fase. Así, un usuario puede realizar el análisis de una aplicación o realizar una simulación detallada sólo en estos representantes, y, combinando estas estadísticas, obtener una representación completa y precisa de la ejecución completa del programa. El representante para cada fase, definido por un cluster de intervalos, se escoge encontrando el intervalo más cercano al centro del cluster, el centroide. El intervalo seleccionado para una fase se denomina punto de simulación para esa fase. El comportamiento de todo el programa es estimado ponderando los resultados de rendimiento de cada punto de simulación por el tamaño y número de intervalos de la fase de la que proviene. *Simpoint* puede reducir significativamente el tiempo de análisis y simulación de una aplicación y permite obtener una caracterización precisa de todo el programa.

Simpoint utiliza como información de entrada lo que se conoce como un vector de bloques básicos, el cual contiene información sobre la cantidad de

veces que es ejecutado cada bloque básico de la aplicación analizada. Para obtener esta información se ha utilizado la herramienta *Pinpoints* [pinb], que viene incluida en la distribución de Pin.

Dado que era necesario simular un entorno multitarea, se utilizó *Simpoint* para que proporcionara sólo un punto de simulación.

Benchmark	171.swim	172.mgrid	173.applu	179.art	183.equake
Simpoint	32	1283	1202	123	104

TABLA 4.1: Simpoints obtenidos para SPEC 2000.

Los puntos de simulación utilizados se presentan en las tablas 4.1 y 4.2. Dichos puntos están expresados en billones de instrucciones.

Benchmark	401.bzip2	403.gcc	410.bwaves	429.mcf	433.milc
Simpoint	284	69	862	153	979
Benchmark	434.zeusmp	435.gromacs	436.cactusADM	437.leslie3d	444.namd
Simpoint	53	299	2224	1558	0
Benchmark	445.gobmk	450.soplex	453.povray	454.calculix	456.hmmer
Simpoint	123	201	105	826	411
Benchmark	458.sjeng	459.gemsFDTD	462.libq	464.h264	465.tonto
Simpoint	381	31	1516	260	1158
Benchmark	470.lbm	471.omnetpp	473.astar	481.wrf	482.sphinx
Simpoint	663	499	346	1043	2061

TABLA 4.2: Simpoints obtenidos para SPEC 2006.

4.4. Benchmarks y configuraciones

Para evaluar las propuestas presentadas en este trabajo se han utilizado los conjuntos de benchmarks SPEC2000 [spea] y SPEC2006 [speb]. Además las pruebas han sido realizadas en un cluster de cómputo, cuyos nodos cuentan con dos Dual Core AMD Opteron Processor 270 que comparten una memoria de 4GB.

La configuración de cache utilizada en las simulaciones cuenta con una cache L1 privada para cada uno de los procesos y cache L2 compartida por todos. Los parámetros de cada una se presentan en la tabla 4.3.

Tamaño Cache L1	32KB
Asociatividad Cache L1	4 vías
Tamaño de Bloque Cache L1	32B
Tamaño Cache L2	8MB
Asociatividad Cache L2	16 vías
Tamaño de Bloque Cache L2	128B

TABLA 4.3: Configuración de la cache.

Para las pruebas se han definido distintas mezclas de aplicaciones de entre los dos conjuntos de benchmarks, que se presentan en la tabla 4.4.

Identificador de Mezcla	**Benchmarks**
MIX01	183, 429, 462, 482
MIX02	181, 183, 429, 482
MIX03	172, 403, 429, 462
MIX04	181, 183, 429, 462
MIX05	171, 181, 470, 482
MIX06	173, 403, 462, 470
MIX07	410, 434, 437, 450
MIX08	444, 456, 471, 473
MIX09	453, 454, 458, 464
MIX10	435, 445, 465, 481

TABLA 4.4: Mezclas de *benchmarks* utilizadas.

4.5. MicroBenchmark

Para la verificación de la implementación de la política peLifo en el simulador, se decidió hacer un *microbenchmark*, que realiza un patrón de acceso a memoria predefinido, del cual se conoce el comportamiento que debe tener la cache al ser realizado.

4.5.1. Configuración del simulador para la evaluación

Se definió una cache con una asociatividad de 16 y un tamaño de línea de 128 bits para la evaluación, pues éste es el valor que se utilizará en las simulaciones posteriores para la cache L2 del sistema.

Para esto, se utiliza sólo un nivel de cache que implementa la política peLifo, aunque dicha política está diseñada para niveles superiores, pues de esta manera se facilita la manipulación del contenido de la cache y no se afectan los resultados que se obtienen. Adicionalmente, fue necesario reducir los tamaños de las épocas definidas en la política, para poder evaluar la política con una cantidad de accesos a memoria manejables. Así, se definió que la época LRU tendría una duración de 64 ingresos a la cache, y la época peLifo de 32 ingresos a los conjuntos de la cache dedicados a la política LRU.

Posteriormente, se crearon trazas de acceso a memoria con un tamaño igual a la asociatividad, que acceden a conjuntos previamente seleccionados, para cubrir los siguientes casos:

- Un conjunto perteneciente al *set dueling* para la política P_1. Se eligió el conjunto 0.
- Un conjunto perteneciente al *set dueling* para la política P_2. Se eligió el conjunto 31.
- Un conjunto perteneciente al *set dueling* para la política P_3. Se eligió el conjunto 10.
- Un conjunto perteneciente al *set dueling* para la política LRU. Se eligió el conjunto 21.

- Un conjunto que no pertenezca a ninguno de los conjuntos de *set dueling*, se ha escogido el conjunto 1.

4.5.2. Verificaciones realizadas

Para cada uno de estos casos se crearon dos patrones de acceso a memoria distintos, lo que hizo posible rellenar el conjunto de la cache y luego forzar reemplazos en el mismo.

Durante la época 1, correspondiente a LRU, se rellenan algunos de los conjuntos y se realizan algunos aciertos para fijar los puntos de escape, llevando a cabo los siguientes accesos:

- Llenado del conjunto 21 (política LRU) usando el primer patrón de acceso a memoria (16 ingresos).
- Accesos al conjunto 21, para generar valores para el *epCounter* y así influir en la escogencia de los puntos de escape.
- Llenado del conjunto 0 (política P_1) usando el primer patrón de acceso (16 ingresos)
- Accesos al conjunto 0, para fijar los puntos de escape.
- Llenado del conjunto 31 (política P_2) usando el primer patrón de acceso (16 ingresos)
- Llenado del conjunto 10 (política P_3) con el primer patrón de acceso (16 ingresos). Cambio de época.

En la época 2, en la cual se ha cambiado a modo de reemplazamiento peLifo, el objetivo fue evaluar la elección de la política ganadora, para esto se generan algunos fallos para forzar que la política seleccionada sea P2, luego se fuerzan algunos fallos de nuevo para que gane la política LRU. Para lograr esto se han seguido los siguientes pasos:

- Llenado del conjunto 1, el cual no pertenece a set dueling.
- Llenado del conjunto 0 (política P_1) con el patrón de acceso 2 para generar fallos e influir en la escogencia de la política ganadora.
- Llenado del conjunto 10 (política P_3) con el patrón de acceso 2 para generar fallos y que esta política no sea elegida como ganadora.
- Se genera un acierto en el conjunto 1, para que al forzar un reemplazo en este bloque, se elimine el elemento siguiente.
- Llenado del conjunto 1 con el patrón de acceso 2 para generar reemplazos. La política ganadora es P_2 pues no ha producido fallos a cache.
- Llenado del conjunto 31 (política P_2) usando el segundo patrón de acceso para generar fallos.
- Llenado del conjunto 1 mediante el primer patrón de acceso para generar reemplazos, ahora la política ganadora debe ser LRU.
- Accesos a conjunto 21 (política LRU) para fijar los puntos de escape.
- Llenado del conjunto 31 con el segundo patrón de acceso (16 ingresos).
- Llenado del conjunto 31 con el primer patrón de acceso (16 ingresos) cambio de época.

En la época 3, se continúa en el modo peLifo, se pretende en este caso forzar un cambio de fase a través del cambio de los puntos de escape, para esto, se realizaron las siguientes operaciones:

- Llenado del conjunto 21 con el segundo patrón de acceso (16 ingresos).
- Accesos al conjunto 21, para fijar los puntos de escape y forzar un cambio de fase.
- Llenado del conjunto 21 con el primer patrón de acceso (16 ingresos). Fin de época.

En esta época se ha cambiado a modo LRU, que era el objetivo. En este punto se detiene el *microbenchmark* pues ya han sido evaluadas las características del algoritmo.

Capítulo 5

Evaluación de resultados

En la primera parte de esta sección se presentan los resultados obtenidos en la simulación de la política peLifo original en comparación a la política LRU en un entorno *singlecore*.

En la segunda parte, se exponen los resultados de las simulaciones de las políticas propuestas en este trabajo, en las cuales se tomarán como base para la comparación el comportamiento presentado por las políticas LRU y peLifo original en un entorno multitarea.

5.1. Entorno *singlecore*

Como se mencionó antes, esta sección muestra el resultado de algunas simulaciones en un entorno *singlecore*, las cuales fueron realizadas con el propósito de comprobar el rendimiento de la política peLifo en estos sistemas. En la figura 5.1 se muestran las tasas de fallos normalizadas respecto a la política LRU de las aplicaciones simuladas. En ésta se comprueba que peLifo genera,

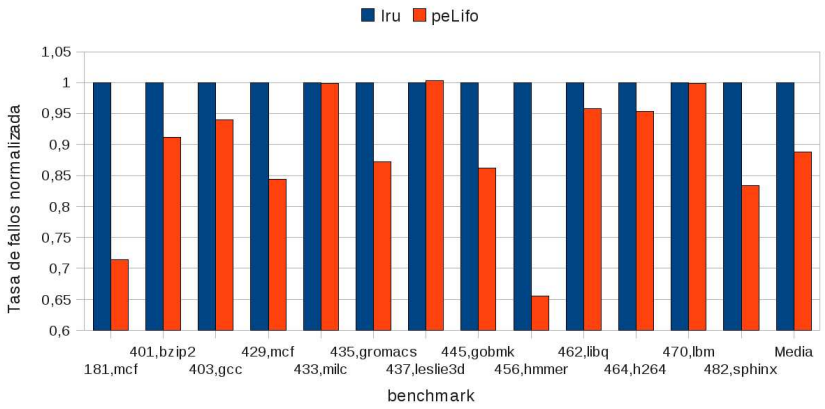

FIGURA 5.1: Tasa de fallos normalizada respecto a LRU.

en media, un 11 % menos de fallos que LRU, siendo esta disminución bastante considerable, similar a la obtenida en el artículo original [Cha09b].

5.2. Entorno *multicore* sin simplificaciones

En esta sección, se presentan los resultados correspondientes a las políticas propuestas en este trabajo: peLifo por listas (peLifo-ls), peLifo proporcional (peLifo-prop) y peLifo proporcional y por listas (peLifo-prop-ls).

Debido a que se trata de un entorno multitarea, los resultados que se obtienen dependen en gran medida de la manera como se entrelazan las ejecuciones de las aplicaciones evaluadas. Este entrelazamiento es aleatorio, lo que ocasiona que para cada simulación se obtenga un resultado diferente. Para enfrentar este problema, se generaron tres trazas por aplicación, y se evaluó cada política con cada una de las trazas, de modo que los resultados fueran comparables. Luego, se calculó una media aritmética con los valores obtenidos para cada una de las trazas.

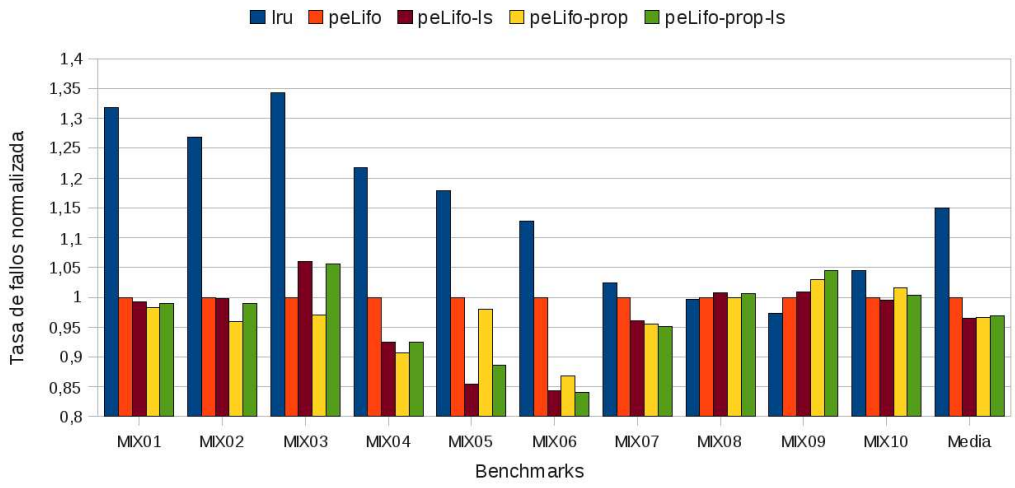

FIGURA 5.2: Tasa de fallos normalizada respecto a peLifo.

Adicionalmente, para situar las tasas de fallos en un mismo rango, se normalizaron los valores medios, anteriormente calculados, respecto a la media de fallos obtenida para la política peLifo original, de modo que se puedan comparar adecuadamente las tasas de fallos obtenidas para las políticas propuestas.

En la figura 5.2, se presenta un diagrama de las tasas de fallos normalizadas respecto a la política peLifo. Además, en la tabla 5.1, se presentan los valores medios de la tasa de fallos normalizada para cada una de las políticas. En esta se puede ver que peLifo mejora significativamente respecto a LRU, aproximadamente 15 % en media.

Política	**Tasa media de fallos**
LRU	1.15
peLifo	1
peLifo-ls	0.96
peLifo-prop	0.97
peLifo-prop-ls	0.97

TABLA 5.1: Tasa media de fallos normalizada respecto a peLifo.

Entre las propuestas realizadas en este trabajo, se destacan los resultados

conseguidos con peLifo-ls, puesto que se logró una mejora en la tasa media de fallos de un 4 % respecto a peLifo. Sin embargo, se debe destacar que la política peLifo-prop logra superar significativamente a peLifo-ls en dos de las mezclas evaluadas, MIX02 y MIX03, y la supera ligeramente en las mezclas MIX07 y MIX08. Esto se puede deber a que, en ocasiones, peLifo-ls penaliza demasiado a alguno de los procesos si éste obtiene una cantidad de aciertos por bloque demasiado baja a lo largo de su ejecución, lo cual hace que se situe con frecuencia en la primera posición de la lista ordenada, haciendo que gran parte de sus bloques se desplacen de la cache y degraden su rendimiento individual, lo que puede conllevar a que se obtenga un rendimiento global menor.

Es de notar que, dado que se trata de la LLC, si se presenta un fallo de cache en este nivel, será necesario pasar la petición del dato a la memoria principal, lo que conlleva un retardo considerable. Es por esto que una reducción en la tasa de fallos puede suponer una mejora significativa en el rendimiento total del sistema.

5.3. Entorno *multicore* con simplificaciones

En esta sección se evaluó el impacto de las simplificaciones propuestas en las políticas de reemplazamiento presentadas, para así definir si las políticas continúan obteniendo una ventaja respecto a la política peLifo original. Las simplificaciones evaluadas fueron: peLifo por listas simplificada (peLifo-ls-simple), peLifo proporcional simplificada (peLifo-prop-simple) y peLifo proporcional y por listas simplificada (peLifo-prop-ls-simple)

En la figura 5.3, se muestran los porcentajes de fallos normalizados de la

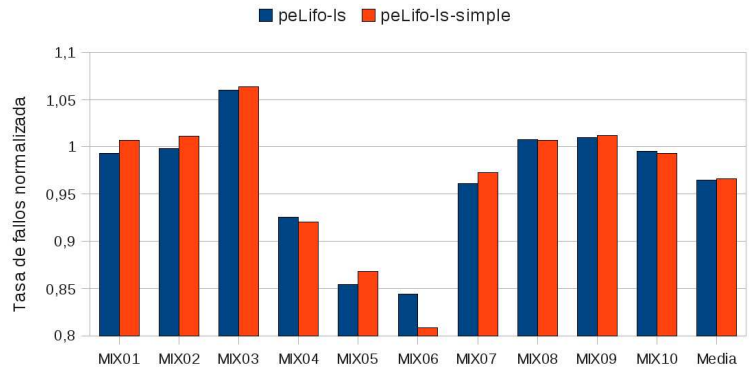

FIGURA 5.3: Tasa de fallos normalizada respecto a peLifo de peLifo-ls.

política peLifo-ls al aplicar las simplificaciones y en ausencia de ellas. En ella se puede ver que la simplificación degrada en media un 1 % el rendimiento de esta propuesta, pasando de 0,96 a 0,97 en la tasa de fallos normalizada.

Por otra parte, la figura 5.4 presenta la tasa de fallos normalizada de la política peLifo-prop con simplificaciones y sin ellas. En este caso, la degradación de rendimiento es mínima a causa de las simplificaciones, obteniendo un valor practicamente igual, manteniendo una media de 0,97.

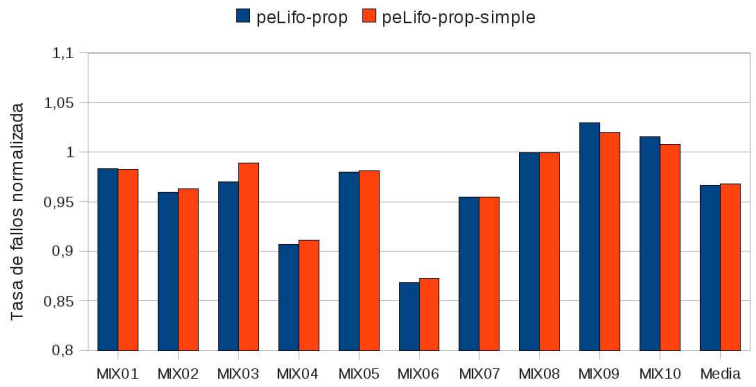

FIGURA 5.4: Tasa de fallos normalizada respecto a peLifo de peLifo-prop.

Por último, la figura 5.5 muestra la tasa de fallos media normalizada para

cada una de las mezclas evaluadas de la política combinada peLifo-prop-ls en ambos casos: con simplificación y sin ella.

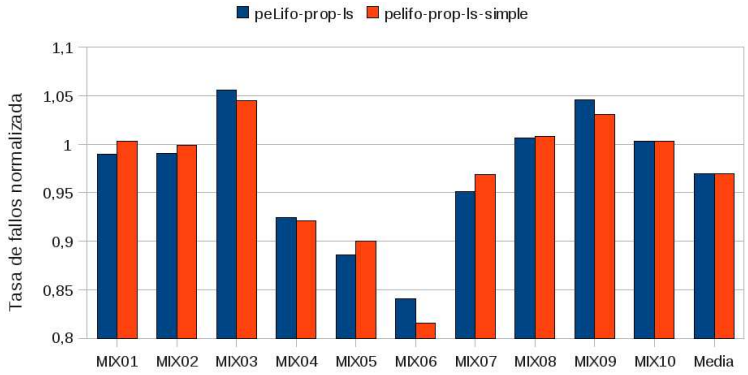

FIGURA 5.5: Tasa de fallos normalizada respecto a peLifo de peLifo-prop-ls.

Al igual que en la política anterior, los resultados se ven muy poco afectados a causa de las simplificaciones, obteniéndose una media para la tasa de fallos normalizada de $0,97$.

De estos resultados se concluye que, aunque las simplificaciones efectuadas en las políticas propuestas degradan un poco los resultados, esta degradación es mínima, por lo que las estrategias propuestas continúan siendo viables.

Capítulo 6

Conclusiones y trabajo futuro

En el presente estudio se ha planteado la necesidad de adaptar las estructuras de los sistemas de cómputo a entornos *multicore*, pues, dado que este tipo de sistemas se hace cada vez más común, es necesario adaptar las diferentes partes del sistema para sacar el máximo provecho de estos. Específicamente, este trabajo ha presentado una propuesta para adaptar la política de reemplazamiento peLifo a entornos *multicore*. Dicha política fue diseñada para ser usada en la LLC, por lo que cualquier mejora obtenida en este nivel de la jerarquía puede implicar una mejora considerable en el rendimiento de un sistema, ya que un fallo ocasionado en este nivel hace que la petición llegue hasta la memoria principal, la cual es mucho más lenta.

Como se mostró en la sección de resultados, es posible obtener aún mayores mejoras usando la política peLifo, sin embargo, esto se lograría a costa de incrementar el hardware necesario para la implementación de la misma. Lo anterior era de esperarse, pues al necesitar cada vez más información sobre los procesos que se ejecutan en un momento determinado, son necesarios más re-

cursos para obtener y procesar estos datos, lo que trae como consecuencia una mayor complejidad en el hardware. Sin embargo, es posible reducir a un mínimo el hardware necesario mediante técnicas de aproximación y simplificación como fue comprobado en este estudio.

En media, se obtuvo un 3 % de mejora, en comparación con la política peLifo original, en todas las modificaciones propuestas aplicando las simplificaciones respectivas.

Como trabajo futuro, se puede extender el concepto de punto de escape a entornos *multicore* asignando puntos de escape individuales a cada uno de los procesos, de modo que se pueda modelar mejor el comportamiento de cada uno de ellos. Durante el desarrollo de éste trabajo, se hicieron algunas pruebas en esta dirección, sin obtener buenos resultados, sin embargo, debería ser posible mejorarlos.

Podrían utilizarse también simplificaciones a la política peLifo original, tema del que existen algunas propuestas, para disminuir aún más el hardware necesario para la implementación de las extensiones presentadas.

Publicaciones realizadas

- S. Sepúlveda, E. Sedano, D. Chaver, F. Castro, L. Piñuel, F. Tirado. "*Simplificación y extensión a un entorno multi-core de la política de reemplazamiento Probabilistic Escape LIFO*". CEDI 2010.

Bibliografía

[Bel66] L. A. Belady. A Study of Replacement Algorithms for Virtual Storage Computers. *IBM Systems Journal*, 5(2):78–101, June 1966.

[Cha09a] M. Chaudhuri. Additional Experiments with Probabilistic Escape LIFO. 2009.

[Cha09b] M. Chaudhuri. Pseudo-LIFO: The Foundation of a New Family of Replacement Policies for Last-Level Caches. *Proceedings of the 42th International Symposium on Microarchitecture*, pages 401–412, 2009.

[HP02] J. Hennessy and D. Patterson. *Computer Architecture: A Quantitative Approach.* Morgan Kaufmann, 3rd edition, 2002.

[JCLJ08] A. Jaleel, R. Cohn, C. Luk, and B. Jacob. CMP$im - A Pin-Based On-The-Fly Multi-Core Cache Simulator. *Fourth Annual Workshop on Modeling, Benchmarking and Simulation*, 2008.

[JHQ+08] A. Jaleel, W. Hasenplaugh, M. Qureshi, J. Sebot, S. Steely, and J. Emer. Adaptive Insertion Policies for Managing Shared Caches. *Proceedings of the 17th International Conference on Parallel*

Architecture and Compilation Techniques, pages 208–219, October 2008.

[JWN07] B. Jacob, D. Wang, and S.Ñg. *Memory Systems: Cache, DRAM, Disk*. Morgan Kaufmann, 1th edition, 2007.

[LCM+05] C. Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. Wallace, V. J. Reddi, and K. Hazelwood. Pin: Building Customized Program Analysis Tools with Dynamic Instrumentation. *Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 190–200, June 2005.

[LFF01] A. Lai, C. Fide, and B. Falsafi. Dead-block Prediction & Dead-block Correlating Prefetchers. *Proceedings of the 28th International Symposium on Computer Architecture*, pages 144–154, June 2001.

[LFHB08] H. Liu, M. Ferdman, J. Huh, and D. Burger. Cache Bursts: A New Approach for Eliminating Dead Blocks and Increasing Cache Efficiency. *Proceedings of the 41th International Symposium on Microarchitecture*, pages 222–233, November 2008.

[MF05] C. McCurdy and C. Fisher. Using Pin as a Memory Reference Generator for Multiprocessor Simulation. *ACM SIGARCH Computer Architecture News*, 33(5):39–44, 2005.

[mul] MultiCacheSim Homepage. http://github.com/blucia0a/MultiCacheSim.

[pel] peLifo Source Code. http://www.cse.iitk.ac.in/ mainakc/pelifo-source.html.

[pina] Pin home page. http://www.pintool.org.

[pinb] Pinpoints Homepage. http://www.cs.virginia.edu/wiki/pin/index.php/PinPoints.

[QJP+07] M. Qureshi, A. Jaleel, Y. Patt, S. Steely, and J. Emer. Adaptive Insertion Policies for High Performance Caching. *Proceedings of the 34th International Symposium on Computer Architecture*, pages 381–391, June 2007.

[QLMP06] M. Qureshi, D. Lynch, O. Mutlu, and Y. Patt. A case for MLP-Aware Cache Replacement. *Proceedings of the 33th International Symposium on Computer Architecture*, 2006.

[QP06] M. Qureshi and Y. Patt. Utility-Based Cache Partitioning: A Low-Overhead, High-Performance, Runtime Mechanism to Partition Shared Caches. *Proceedings of the 39th International Symposium on Microarchitecture*, pages 423–432, 2006.

[ses] SESC simulator home page. http://sesc.sourceforge.net.

[sim] Simpoint home page. http://cseweb.ucsd.edu/ calder/simpoint.

[spea] SPEC 2000 home page. http://www.spec.org/cpu2000.

[speb] SPEC 2006 home page. http://www.spec.org/cpu2006.

[SRD04] G. Suh, L. Rudolph, and S. Devadas. Dynamic Partitioning of Shared Cache Memory. *Journal of Supercomputing*, pages 7–26, June 2004.